# Hoofdstuk 1

# C# in een notedop

In de volgende hoofdstukken zullen we een aantal aspecten van het .NET-platform behandelen. We maken hierbij gebruik van de object-georiënteerde programmeertaal C# (C sharp uitgesproken). Dit hoofdstuk bevat een beknopt overzicht van C# en de gelijkenissen en verschillen met Java.

## 1.1 Gegevenstypes

De declaratie en toekenning van een variabele heeft in C# dezelfde syntax als in Java of C++.

```
type naamVariabele;
naamVariabele = waarde;
```

of verkort

```
type naamVariabele = waarde;
```

C# heeft twee soorten gegevenstypes: *waarde-types* en *referentie-types*. Het verschil tussen beide types is dat variabelen van het eerste type hun data bevatten en variabelen van het tweede type een verwijzing naar een object.

### 1.1.1 Waarde-types

Voorgedefinieerde waarde-types zijn onder andere: *int*, *long*, *float*, *double*, *bool*, *char*, ... De volgende tabel geeft een beknopt overzicht van deze types.

| | |
|---|---|
| *int* | Geheel getal (32 bits) |
| *long* | Geheel getal (64 bits) |
| *float* | Vlottendekommagetallen met enkelvoudige precisie |
| *double* | Vlottendekommagetallen met dubbele precisie |
| *bool* | Logisch type kan waarde *true* of *false* aannemen |
| *char* | 16 bits unicode letterteken |

Met een *struct* (zie §1.6)- of *enum* (zie §1.4)-declaratie is het mogelijk om nieuwe waarde-types te definiëren.

**Voorbeelden**

```
int geheleWaarde = 257;
long langeWaarde = 2200000;
float fWaarde = 2.15 ;
double dWaarde = 4.5789 ;
bool gelijk = false;
char letter = 'k';
```

### 1.1.2 Referentie-types

In C# zijn er twee voorgedefinieerde referentie-types: *object* en *string*. Het type *object* is het basistype voor alle andere types, ook voor de waarde-types. Om unicode-strings voor te stellen wordt het type *string* gebruikt. Variabelen van het type *string* zijn *immutable*. Dit betekent dat de waarde van een *string*-variabele niet veranderd kan worden. Om een string aan te passen wordt een nieuw object aangemaakt.

Nieuwe referentie-types worden gecreëerd met behulp van *klasse* (zie §1.5)-, *interface* (zie §1.8)- en *delegate*-declaraties.

**Voorbeelden**

```
string naam = "Pieter-Jan";
object iets = null;
```

Merk op dat het sleutelwoord *null* gebruikt wordt voor een lege verwijzing.

**Tabellen**

Een één-dimensionale tabel wordt als volgt gedeclareerd

```
type[] tabel;
```

waarbij *type* het type voorstelt van de elementen in de tabel. De volgende pseudocode illustreert het aanmaken van een tabel van lengte *N*. *N* is een positieve gehele waarde en *type* is het type van de elementen in de tabel.

```
type[] tabel = new type[N];
```

Enkele voorbeelden:

```
int[] gehelen = new int[3];
gehelen[0] = 15;
gehelen[1] = -231;
gehelen[2] = 3;

int[] evenCijfers = new int[5]{0,2,4,6,8};

int[] onevenCijfers = {1,3,5,7,9};
```

De opdracht *int[] onevenCijfers = {1,3,5,7,9};* is een verkorte notatie voor *int[] onevenCijfers = new int[5]{1,3,5,7,9};*.

De volgende voorbeelden illustreren het gebruik van meerdimensionale tabellen.

```
int[][] getallen = new int[3][];
getallen[0] = new int[1]{15};
getallen[1] = new int[4]{-231,48,97,-2};
getallen[2] = new int[2]{3,56};

char[][][] letters = new char[2][][];
letters[0] = new char[3][];
letters[0][0] = new char[3]{'d','i','t'};
letters[0][1] = new char[2]{'i','s'};
letters[0][2] = new char[5]{'g','r','o','e','n'};
letters[1] = new char[1][];
letters[1][0] = new char[3];
letters[1][0][0] = 'a';
letters[1][0][1] = 'x';
letters[1][0][2] = 'p';
```

De syntax voor rechthoekige tabellen is iets eenvoudiger.

```
double [,] coord2D = {{1.3,-5.2},{1.0,4.5},{0.284,-1.003}};
```

```
double [,,] coord3D = new double[4,3,2];
for (int i = 0; i < 4; i++)
    for (int j = 0; j < 3; j++)
        for (int k = 0; k < 2; k++)
            coord3D[i,j,k] = 1;
```

### 1.1.3 Opmerkingen

Elk voorgedefinieerd type is een verkorte notatie van een *struct* (zie §1.6) uit de basisbibliotheek. De sleutelwoorden *int*, respectievelijk *string*, zijn bijvoorbeeld een verkorte vorm van *System.Int32*, respectievelijk *System.String*. De standaard programmeerstijl voor C# verkiest het gebruik van de verkorte notatie.

Elk type is afgeleid van het type *object*. Dit betekent dat de methodes van *object* methodes zijn voor alle andere types, ook voor waarde-types. De volgende uitdrukkingen zijn dus correct.

```
int geheleWaarde = 893;
string stringWaarde = geheleWaarde.ToString();
stringWaarde = 893.ToString();
```

### 1.1.4 Conversies

Voor de voorgedefinieerde waarde-types van C# bestaan conversies die een variabele van het ene type omzetten in een variabele van het andere type. Indien deze conversie steeds op een veilige manier kan gebeuren (bv. van *int* naar *long*) dan noemen we dit een *impliciete* conversie, in het andere geval spreken we van een *expliciete* conversie. In de syntax van C# hoef je een impliciete conversie niet aan te duiden, een expliciete conversie daarentegen duid je aan met een cast-uitdrukking.

Het volgende voorbeeld illustreert het gebruik van een impliciete en een expliciete conversie tussen een variabele van het type *int* en een variabele van het type *long*.

```
int geheleWaarde = 257;
long langeWaarde = geheleWaarde; // impliciete conversie

langeWaarde = 2200000;
geheleWaarde = (int) langeWaarde; // expliciete conversie
```

Het converteren van een variabele van een waarde-type naar een variabele van het type *object* en omgekeerd wordt respectievelijk *boxing* en *unboxing* genoemd.

```
int geheleWaarde = 257;
object objWaarde = geheleWaarde; // boxing
int geheel = (int) objWaarde; // unboxing
```

Merk op dat boxing een veilige conversie is en dus geen cast-uitdrukking nodig heeft.

Bij boxing wordt de waarde van de variabele gekopieerd in een nieuw gecreëerd object. Veranderingen aan dit object impliceren dus *geen* veranderingen aan de inhoud van de oorspronkelijke variabele.

```
int geheleWaarde = 257;
object objWaarde = geheleWaarde;
objWaarde = 342;
```

Na het uitvoeren van bovenstaande code is de waarde van *geheleWaarde* nog steeds *257*.

Boxing en unboxing maakt het mogelijk om methodes waarbij het argument of de waarde een object zijn, te gebruiken voor waardetypes. Een methode *Methode* met de volgende signatuur

```
public object Methode(object o)
```

zou bijvoorbeeld als volgt gebruikt kunnen worden:

```
int geheleWaarde, geheel;
geheleWaarde = -3654;
geheel = (int) Methode(geheleWaarde);
```

De laatste uitdrukking maakt dus zowel gebruik van boxing (om *geheleWaarde* om te vormen naar een object) als unboxing (om het resultaat van *Methode* te converteren naar een *int*-waarde).

## 1.2 Uitdrukkingen

De operatoren die je kan gebruiken om uitdrukkingen te maken zijn gelijkaardig aan die in C++ of Java. De tabel geeft een overzicht van een aantal bewerkingen. De operatoren zijn gerangschikt volgens prioriteit. Diegene die eerst voorkomen in de tabel hebben de hoogste prioriteit.

| | |
|---|---|
| ! | negatie |
| $*$ | vermenigvuldiging |
| $/$ | deling |
| $\%$ | rest bij deling |
| $+$ | optelling |
| $-$ | aftrekking |
| $<$ | kleiner dan |
| $>$ | groter dan |
| $<=$ | kleiner dan of gelijk aan |
| $>=$ | groter dan of gelijk aan |
| $==$ | gelijk aan |
| $!=$ | niet gelijk aan |
| $\&\&$ | en |
| $\|\|$ | of |
| $=$ | toekenning |
| $*=$ | vermeningvuldig en toekenning |
| $/=$ | deling en toekenning |
| $\%=$ | rest bij deling en toekenning |
| $+=$ | optelling en toekenning |
| $-=$ | verschil en toekenning |

Het resultaat van deze operatoren hangt af van het type van de gebruikte argumenten. Zo zal het resultaat van *4/5* de waarde *0* zijn, terwijl het resultaat van *4.0/5.0* de waarde *0.8* is.

### 1.2.1 Vergelijken van strings

In C# worden twee *strings* als gelijk beschouwd indien ze evenlang zijn en op elke positie hetzelfde karakter hebben staan, of als ze beide *null* zijn. Twee variabelen van het type *object* zijn gelijk als ze naar hetzelfde object verwijzen.

```
string woord1 = "Test";
string woord2 = string.Copy(woord1);
bool gelijkString = woord1==woord2; // true
bool gelijkObject = (object)woord1==(object)woord2; // false
```

In de bovenstaande code heeft de logische variabele *gelijkString* de waarde *true* en de logische variabele *gelijkObject* de waarde *false*, omdat de twee variabelen hetzelfde woord voorstellen, maar niet naar hetzelfde object verwijzen.

## 1.3 Programmastructuren

In deze sectie geven we een kort overzicht in pseudocode van een aantal programmastructuren in C#. De meeste van deze structuren zijn hetzelfde als in C++ of Java.

### 1.3.1 Selectie

De *if-else*-opdracht realiseert een selectie op basis van een logische uitdrukking.

```
if (voorwaarde) {
    ... // uit te voeren opdrachten als 'voorwaarde' waar is
} else {
    ... // uit te voeren opdrachten als 'voorwaarde' onwaar is
}
```

Hierbij is *voorwaarde* een logische uitdrukking van het type *bool*. De *else*-clausule mag weggelaten worden. Indien de uit te voeren opdrachten maar één opdracht omvat mogen de accolades weggelaten worden. Indien het *else*-blok enkel uit een *if* - of een *if-else*-opdracht bestaat gebruiken we de volgende verkorte notatie.

```
if (voorwaarde1) {
    ... // uit te voeren opdrachten als 'voorwaarde1' waar is
} else if (voorwaarde2) {
    ... // uit te voeren opdrachten als 'voorwaarde2' waar is
} else {
    ... /* uit te voeren opdrachten als 'voorwaarde1' en
         'voorwaarde2' onwaar zijn */
}
```

De *switch*-opdracht maakt een selectie op basis van de waarde van een uitdrukking mogelijk.

```
switch (uitdrukking) {
    case waarde1:
        ... /* uit te voeren opdrachten
                als de waarde van 'uitdrukking' 'waarde1' is */
       break;
    case waarde2:
        ... /* uit te voeren opdrachten
                als de waarde van 'uitdrukking' 'waarde2' is */
        break;

    ... // mogelijke andere case-structuren

    default:
        ... /* uit te voeren opdrachten als de waarde van
                'uitdrukking' geen enkele van de
           bovenstaande waarden ('waarde1', 'waarde2', ...) is */
        break;
  }
```

Het type van *uitdrukking* moet impliciet converteerbaar zijn naar een beperkt aantal voorgedefinieerde types (o.a. *int*, *long*, *char* en *string*) of naar een *enum* (zie §1.4)-type. De waarden *waarde1*, *waarde2*,

... zijn constante uitdrukkingen van een type dat impliciet converteerbaar moet zijn naar het type van *uitdrukking*. De *default*-optie mag weggelaten worden.

### 1.3.2 Herhaling

Herhaling afhankelijk van de waarde van een logische uitdrukking (*voorwaarde* in de pseudocode) wordt gerealiseerd door een *while*- opdracht.

```
while (voorwaarde) {
    ... // uit te voeren opdrachten zolang 'voorwaarde' waar is
}
```

Een herhaling die een welbepaald aantal keer uitgevoerd moet worden kan gebruik maken van een *for*-opdracht.

```
for (initialisatie; voorwaarde; iteratie) {
    ... // uit te voeren opdrachten zolang 'voorwaarde' waar is
}
```

In deze structuur zijn *initialisatie* en *iteratie* commando's en is *voorwaarde* een logische uitdrukking.

Om voor een aantal elementen van een verzameling een reeks opdrachten uit te voeren kan je gebruik maken van de *foreach*-opdracht.

```
foreach (type element in verzameling) {
    ... /* uit te voeren opdrachten voor elk element 'element'
             in 'verzameling' */
}
```

Hierbij is *type* het type van de variabele *element*. Het type van de variabele *verzameling* moet de interface *System.Collections.IEnumerable* implementeren. Bovendien mag de verwijzing (pointer) die de iteratie-variabele *element* bevat niet gewijzigd worden in de *foreach*-lus.

### 1.3.3 Uitzonderingen

Het gooien van een uitzondering verloopt als volgt:

```
throw uitzondering;
```

Het type van de variabele *uitzondering* moet *System.Exception* of een afgeleide klasse van *System.Exception* zijn.

Met behulp van een *try-catch-finally*-structuur kan je uitzonderingen opvangen.

```
try {
    ...
} catch (Exception1 e1) {
    ...
} catch (Exception2 e2) {
    ...

... // eventueel andere catch-blokken

} finally {
    ...
}
```

*Exception1* en *Exception2* zijn de types van de variabelen *e1* en *e2*. Beide types zijn *System.Exception* of een hiervan afgeleid type. De *catch*-blokken worden in volgorde van voorkomen afgehandeld. Dit betekent dat de types van de uitzonderingen geen afgeleide klassen mogen zijn van die types die vroeger in een *catch*-blok voorkwamen. De *catch*-blokken en het *finally*-blok zijn beiden optioneel, slechts één van beiden moet voorkomen.

## 1.4 Enum

Het *enum*-type wordt gebruikt om een type te declareren dat bestaat uit verwante symbolische constanten. De declaratie van een *enum*-type heeft de volgende vorm:

```
enum NaamType {SymCst1, SymCst2, ..., SymCstn}
```

*NaamType* is de naam van het nieuwe type, *SymCst1*, *SymCst2*, ... de symbolische constanten waaruit het type bestaat. Het onderliggende type van zo'n *enum*-declaratie is *int*. Dit betekent dat elke symbolische constante een geassocieerde gehele waarde heeft. Met de uitdrukking *NaamType.SymCsti* kan je de waarde van de symbolische constante verkrijgen. Standaard krijgt de eerste symbolische constante de waarde 0, de tweede waarde 1, enzovoort. Het is ook mogelijk om dit zelf in te stellen.

Een aantal voorbeelden:

```
enum Vorm {Cirkel, Vierkant, Rechthoek, Driehoek}

enum Dag {Maandag, Dinsdag, Woensdag, Donderdag, Vrijdag,
    Zaterdag, Zondag}
```

```
enum Kleur {Geel, Rood, Blauw, Groen}
```

Het *enum*-type kan in combinatie met een *switch*-opdracht gebruikt worden, indien uit een vast aantal mogelijkheden gekozen moet worden.

```
Vorm mijnFiguur = Vorm.Driehoek;
switch (mijnFiguur) {
    case Vorm.Cirkel:
        ... // Teken cirkel
        break;
    case Vorm.Vierkant:
        ... // Teken vierkant
        break;
    case Vorm.Rechthoek:
        ... // Teken rechthoek
        break;
    case Vorm.Driehoek:
        ... // Teken driehoek
        break;
}
```

Ook in een *foreach*-opdracht kan een *enum*-type gebruikt worden. Het onderstaande stukje code illustreert ook dat het onderliggend type van een *enum*-type een *int* is.

```
Vorm[] verzameling = {Vorm.Vierkant, Vorm.Driehoek};
foreach (Vorm figuur in verzameling) {
    System.Console.WriteLine(figuur);
    System.Console.WriteLine((int) figuur);
}
```

De uitvoer van dit stukje code is

```
Vierkant
1
Driehoek
3
```

## 1.5 Klassen

Met behulp van klassedeclaraties kan je nieuwe referentietypes definiëren. Klassen kunnen afgeleid zijn van andere klassen (§1.12) of interfaces (zie §1.8) implementeren. Mogelijke leden van een klasse zijn

- constanten
- velden
- constructoren
- methodes (zie §1.7)
- eigenschappen (zie §1.9)
- indexen (zie §1.10)
- inwendige klassen
- gebeurtenissen
- operatoren
- destructoren

De bereikbaarheid van deze leden wordt bepaald door een sleutelwoord dat de toegang bepaalt. Mogelijke waarden zijn:

| | |
|---|---|
| *public* | Leden van alle klassen hebben toegang. |
| *protected* | Enkel toegang voor leden van de klasse die het lid bevat en daarvan afgeleide klassen. |
| *internal* | Toegang beperkt tot de applicatie waartoe de klasse behoort. |
| *protected internal* | Combinatie van de twee voorgaande. |
| *private* | De toegang is beperkt tot de klasse die het lid bevat. Dit is de standaardtoegang als er geen toegang gespecifieerd is. |

### 1.5.1 Declaratie van een klasse

Een klassedeclaratie heeft de volgende structuur. Hierbij kan *toegang* onder andere de waarden *private*, *protected*, *internal* en *public* aannemen.

```
[toegang] class NaamKlasse {
  ... // declaratie van de leden
}
```

### 1.5.2 Declaratie van een constante

De syntax van een constantedeclaratie is de volgende:

```
[toegang] const type naamConstante;
```

*type* is het type van de constante, *naamConstante* de naam van de constante.

### 1.5.3 Declaratie van een veld

De syntax van een velddeclaratie is de volgende:

```
[toegang] [static] [readonly] type naamVeld;
```

*type* is het type van het veld, *naamVeld* de naam van het veld. Het sleutelwoord *static* geeft aan of het al dan niet om een statisch veld gaat. Statische leden zijn leden van de klasse en niet van de instantiaties van de klassen (objecten). In C# kan je de statische leden van een klassen niet oproepen op een instantiatie van de klasse. Velden die *readonly* gedeclareerd zijn moeten geïntialiseerd worden bij declaratie of in de constructor. Nadien kunnen ze niet meer gewijzigd worden.

### 1.5.4 Declaratie van een constructor

Een constructor definieert alle opdrachten die nodig zijn om een object van het type van de klasse aan te maken en te initialiseren.

```
[toegang] NaamKlasse(argumentenlijst);
```

De constructor heeft dezelfde naam als de klasse. Een constructor kan een argumentenlijst hebben. Een argumentenlijst heeft de volgende syntax:

```
typeParameter1 param1, typeParameter2 param2, ... ,
    typeParameterN paramN
```

Hierbij is *typeParameteri* het type van de i-de parameter en *parami* de naam.

## 1.6 Struct

Met behulp van *struct*s kan je in C# nieuwe waardetypes declareren. Structs kunnen dezelfde leden hebben als klassen en kunnen interfaces implementeren. Ze kunnen echter niet afgeleid worden of afgeleid zijn van een andere struct. De syntax van een *struct*:

```
struct NaamStruct {
   ... // declaratie leden
}
```

Een struct kan geen constructor zonder argumentenlijst hebben. Deze is reeds voorzien en geeft alle leden hun standaardstatus. Bijvoorbeeld 0 voor een *int*.

## 1.7 Methodes

De syntax van een methodedeclaratie is

```
[toegang] [static] ReturnType NaamMethode(parameterLijst) {
    .. // uit te voeren opdrachten
}
```

Net zoals bij statische velden, zijn statische methodes, methodes van de klasse en niet van de instantiaties (objecten). *ReturnType* is het type van het resultaat van de methode of *void* als de methode een procedure is. De parameterLijst is optioneel en heeft de volgende vorm:

```
typeParameter1 param1, typeParameter2 param2, ... ,
    typeParameterN paramN
```

Hierbij is *typeParameteri* het type van de i-de parameter en *parami* de naam.

### 1.7.1 Referentie- en uitvoerparameters

Bij parameters van het type referentietype wordt steeds de referentie naar het object meegegeven aan de methode. De referentie kan niet gewijzigd worden door de methode, maar het object waarnaar de referentie verwijst kan dat wel. Bij parameters van het type waardetype wordt de waarde van de oorspronkelijke variabele gekopieerd en meegegeven aan de methode. De methode kan de oorspronkelijke waarde dus niet veranderen. Om dit toch te kunnen verwezenlijken moet je gebruik maken van referentie- of uitvoerparameters.

Een referentieparameter wordt gekenmerkt door het sleutelwoord *ref* voor het type in de parameterlijst en voor de naam van de variabele in de methode-oproep. We illustreren dit aan de hand van een voorbeeld. De methode *reset* stelt de meegegeven gehele parameter gelijk aan 0.

```
void reset(ref int getal) {
    getal = 0;
}
```

Na het uitvoeren van de volgende opdrachten zal de variabele *getal* de waarde 0 bevatten.

```
int getal = 10;
reset(ref getal);
```

Een uitvoerparameter is een speciale referentieparameter en wordt gekenmerkt door het sleutelwoord *out*. Hij hoeft niet geïnitialiseerd te worden voor de methode-oproep. Dit wordt geïllustreerd in het volgend voorbeeld. De methode met een uitvoerparameter:

```
void init(out int waarde) {
    waarde = 0;
}
```

Dit stukje code toont het gebruik van de bovenstaande methode:

```
int waarde;
init(out waarde);
```

### 1.7.2 Tabel van parameters

In C# is het mogelijk om een methode te definiëren die nul of meer argumenten van hetzelfde type heeft. Dit kan op de volgende manier:

```
  type1 naamMethode (params type2[] naamParam) {
  }
```

Hierbij is *type1* het gegevenstype van de waarde van de methode, *naamMethode* de naam van de methode, *type2* het gegevenstype van de argumenten en *params* een sleutelwoord van C#.

We kunnen nu een functie *Som* maken die de som van een willekeurig aantal reële getallen berekent.

```
double Som(params double[] elementen) {
    double som = 0.0;
    for (int i = 0; i < elementen.Length; i++)
        som += elementen[i];
    return som;
}
```

Deze functie kan je nu op de volgende manier gebruiken:

```
double resultaat;
resultaat = Som();
resultaat = Som(1.0);
resultaat = Som(2.3,2.6);
resultaat = Som(new double[] {1,3.0,4.2});
```

Merk op dat je zowel meerdere argumenten kan hebben als één argument dat een tabel is.

Een methode kan maar één tabel van parameters hebben. Indien de methode nog andere argumenten heeft, dan moet de parameter-tabel het meest rechtste argument zijn.

## 1.8 Interfaces

Een interface is een soort contract. Dit betekent dat een klasse of struct die een interface implementeert, alle leden van de interface moet uitwerken.

Een interface is een manier om naar een object te kijken. We hoeven enkel die interfaces van het object te kennen die we nodig hebben en niet het eigenlijke type (klasse) van het object.

De syntax van een interface-declaratie is de volgende:

```
[toegang] interface INaamInterface {
    ... // leden interface
}
```

De standaard programmeerstijl voor C# adviseert om de naam van een interface altijd te beginnen met een I (in het voorbeeld *INaamInterface*). Een interface kan methodes (zie §1.7), eigenschappen (zie §1.9), indexers (zie §1.10) en gebeurtenissen bevatten. De interface bevat hiervan enkel de signatuur, de klassen die de interface implementeren bevatten de eigenlijke uitwerking.

In de klasse-, struct- of interfacedeclaratie moet aangegeven worden welke interfaces de klasse implementeert. In het volgend voorbeeld implementeert de klasse *MijnKlasse* de twee interfaces *IInterface1* en *IInterface2*.

```
class MijnKlasse : IInterface1, Interface2 {
    ... // leden, implementeren leden interfaces
}
```

## 1.9 Eigenschappen

*Eigenschappen* zijn uitbreidingen van velden en beschrijven kenmerken van een object of klasse. Integenstelling tot een veld hoeft een eigenschap niet verbonden te zijn met een geheugenlocatie. Een

eigenschap heeft een *get*- en een *set*-methode die de opdrachten bevatten die uitgevoerd moeten worden bij het lezen of veranderen van een eigenschap. De syntax om de waarde van een eigenschap op te halen of te veranderen is dezelfde als bij velden. Maar in plaats van de inhoud van een geheugenlocatie te lezen of aan te passen, wordt de respectievelijke *get*- of *set*-methode uitgevoerd.

De declaratie-syntax van een eigenschap is de volgende:

```
[toegang] type NaamEigenschap {
    get {
        ... /* opdrachten uit te voeren bij het
            lezen van de eigenschap*/
    }
    set {
        ... /* opdrachten uit te voeren bij het
            veranderen van de eigenschap*/
    }
}
```

*type* is het gegevenstype van de eigenschap, *NaamEigenschap* de naam. Indien de *set*-declaratie weggelaten wordt dan is de eigenschap *read-only*. De *get*-declaratie moet een *return*-opdracht bevatten die de waarde van de eigenschap weergeeft.

De waarde die in de *set*-methode aan de eigenschap moet toegekend worden, wordt weergegeven door een impliciete variabele *value*. Dit wordt geïllustreerd in het volgende voorbeeld. De onderstaande klasse *Persoon* heeft een publieke eigenschap *Leeftijd*.

```
public class Persoon {
    private int leeftijd;

    public Persoon(int leeftijd) {
        this.leeftijd = leeftijd;
    }

    public int Leeftijd {
        get {
            return this.leeftijd;
        }

        set {
            if (value >= 0 && value <= 150)
                this.leeftijd = value;
            else
                 throw new Exception("Onmogelijke leeftijd");
        }
    }
}
```

We kunnen nu de eigenschap *Leeftijd* gebruiken alsof het een veld was.

```
// aanmaken van een student
Persoon student = new Persoon(20);
// leeftijd uitschrijven
System.Console.WriteLine(student.Leeftijd);
// leeftijd aanpassen en opnieuw uitschrijven
student.Leeftijd = 21;
System.Console.WriteLine(student.Leeftijd);
```

Een eigenschap hoeft niet verbonden te zijn met een privaat veld. Om dit te illustreren voegen we een *read-only*-eigenschap *Volwassen* toe aan de bovenstaande klasse *Persoon*.

```
public bool Volwassen {
    get {
        if (this.leeftijd < 18)
            return false;
        else
            return true;
    }
}
```

## 1.10 Indexers

In C# is het mogelijk om klassen te creëren die zich gedragen als virtuele tabellen of hashtabellen. Dit wordt gerealiseerd met behulp van *indexers*. Je kan deze klassen gebruiken alsof het tabellen zijn. Dit betekent dat als je een klasse MijnCol hebt met een indexer van het type *object* en een default-constructor, je de volgende opdrachten kan uitvoeren.

```
MijnCol objMetInd = new MijnCol();
object obj1, obj2 = new object();
obj1 = objMetInd[1];
objMetInd[5] = obj2;
```

Merk op dat *objMetInd* helemaal geen tabel is. Bovendien is het ook mogelijk om als index (i.e. het argument van de operator []) niet enkel natuurlijke getallen te gebruiken, maar ook *strings*.

De syntax van een indexer-declaratie bestaat uit een *get*- en een *set*-deel. Het *get*-gedeelte bestaat uit de opdrachten die uitgevoerd worden als men de waarde van *objMetInd[index]* moet bepalen. Dit stuk bevat bijgevolg dus een *return*-opdracht. De opdrachten die *objMetInd[index]* veranderen worden opgenomen in het *set*-blok. De waarde die toegekend moet worden wordt voorgesteld door de impliciete variabele *value*. De syntax in pseudocode:

```
[toegang] type this[int index] {
    get {
```

```
        ...
    }

    set {
        ...
    }
}

[toegang] type this[string sleutel] {
    get {
        ...
    }

    set {
        ...
    }
}
```

Hierbij stelt *type* het gegevenstype voor. Een klasse kan beide type indexers hebben of maar één van de twee. In deze cursus zullen we zelf niet veel klassen met een indexer aanmaken, maar vooral gebruik maken van klassen uit de Klassenbibliotheek van het .NET-Framework (.NET Framework Class Library). In het volgend voorbeeld gebruiken we de klasse *NameValueCollection* uit de bibliotheek *System.Collections.Specialized*. In de methode *VulOp* worden er drie *strings* toegevoegd aan een *NameValueCollection*. Deze strings worden geassocieerd met de sleutelwoorden 'naam', 'voornaam' en 'onderwerp'.

```
private static void VulOp(NameValueCollection geg,
    string naam,
    string voornaam, string onderwerp) {
geg["naam"] = naam;
geg["voornaam"] = voornaam;
geg["onderwerp"] = onderwerp;
}
```

In de methode *Schrijf* worden alle objecten van een *NameValueCollection* uitgeschreven en vervolgens ook de objecten geassocieerd met de sleutelwoorden 'naam', 'voornaam' en 'onderwerp'.

```
private static void Schrijf(NameValueCollection geg) {
    for (int i = 0; i < geg.Count; i++)
        System.Console.WriteLine(geg[i]);
    System.Console.WriteLine();
    System.Console.WriteLine(geg["naam"]);
    System.Console.WriteLine(geg["voornaam"]);
    System.Console.WriteLine(geg["onderwerp"]);
    System.Console.WriteLine();
}
```

De uitvoer van het volgend programma

```
public static void Main() {
    NameValueCollection geg = new NameValueCollection();

    VulOp(geg,"De Ridder","Jan","Muziek");
    Schrijf(geg);

    VulOp(geg,"Peeters","Piet","Sport");
    Schrijf(geg);
}
```

is dan

```
De Ridder
Jan
Muziek

De Ridder
Jan
Muziek

Peeters
Piet
Sport

Peeters
Piet
Sport
```

## 1.11 Namespaces

Net zoals packages in Java, worden *namespaces* in C# gebruikt om stukken code (klassen, interfaces, enums, structs, ...) te structureren. Samenhangende klassen, interfaces, enums, structs, ... behoren tot dezelfde *namespace*.

In het volgend voorbeeld wordt een klasse *Test* aangemaakt die behoort tot de namespace *Be.Hogent.Iii*. Dit kan op twee manieren.

```
namespace Be {
    namespace Hogent {
        namespace Iii {
            public class Test { ... }
        }
    }
}
```

of

```
namespace Be.Hogent.Iii {
    public class Test { ... }
}
```

Merk op dat de klasse *Test* niet tot een directorystructuur "Be/Hogent/Iii"moet behoren.

Om de klassen, interfaces, ... van de namespace *Be.Hogent.Iii* te gebruiken zonder expliciet de volledige naam te gebruiken, voeg je boven de klasse-definitie de volgende opdracht toe.

```
using Be.Hogent.Iii;
```

Je kan ook een *alias* maken voor een klasse die je vaak gebruikt. Bijvoorbeeld,

```
using Test = Be.Hogent.Iii.Test;
```

Nu kan je in een programma *Test* gebruiken in plaats van *Be.Hogent.Iii.Test*.

## 1.12 Afgeleide klassen en abstracte klassen

### 1.12.1 Afgeleide klassen

C# ondersteunt enkelvoudige overerving. De klasse *object* is de klasse waarvan alle andere klassen afgeleid zijn. In het volgend voorbeeld is *MijnKlasse* een afgeleide klasse van de klasse *BasisKlasse* met één methode *MethodeBK*. *MijnKlasse* neemt de methode *MethodeBK* over van *BasisKlasse* en heeft een eigen methode *MethodeMK*.

```
public class BasisKlasse {
    public void MethodeBK() {
        System.Console.WriteLine("BasisKlasse.MethodeBK");
    }
}

    public class MijnKlasse:BasisKlasse {
        public void MethodeMK() {
            System.Console.WriteLine("MijnKlasse.MethodeMK");
        }
    }
```

**Illustratie gebruik**

De uitvoer van de volgende opdrachten

```
MijnKlasse mijnObject = new MijnKlasse();
mijnObject.MethodeBK();
mijnObject.MethodeMK();

BasisKlasse basisObject = mijnObject;
basisObject.MethodeBK();

basisObject = new BasisKlasse();
basisObject.MethodeBK();
```

is dan

```
BasisKlasse.MethodeBK
MijnKlasse.MethodeMK
BasisKlasse.MethodeBK
BasisKlasse.MethodeBK
```

Net zoals bij het implementeren van interfaces, wordt het afleiden van een klasse aangeduid met een dubbel punt. Indien een klasse afgeleid is van een andere klasse en één of meerdere interfaces implementeert, dan moeten na de dubbelpunt eerst de klasse en vervolgens de interfaces vermeld worden.

```
public class MijnKlasse : BasisKlasse, IInterface1, IInterface2 {
      ...
}
```

In het bovenstaande voorbeeld kan *MijnKlasse* de methode *MethodeBK* niet overschrijven. Opdat dit mogelijk zou zijn moet de methode *MethodeBK* van *BasisKlasse* de prefix *virtual* hebben. Bovendien moet je in de klasse *MijnKlasse* expliciet vermelden dat je de *MethodeBK* overschrijft. Dit gebeurt met de prefix *override*. Merk op dat in de code het sleutelwoord *base* gebruikt wordt om de klasse aan te duiden waarvan afgeleid wordt. Een afgeleide klasse kan een 'virtuele' methode overschrijven, maar hoeft dit niet te doen.

```
public class BasisKlasse {
    public virtual void MethodeBK() {
        System.Console.WriteLine("BasisKlasse.MethodeBK");
    }
}

public class MijnKlasse:BasisKlasse {
```

```
    public override void MethodeBK() {
         base.MethodeBK();
         System.Console.WriteLine("MijnKlasse.MethodeBK");
    }

    public void MethodeMK() {
        System.Console.WriteLine("MijnKlasse.MethodeMK");
    }
}
```

De uitvoer van de bovenstaande opdrachten (zie §1.12.1) is nu

```
BasisKlasse.MethodeBK
MijnKlasse.MethodeBK
MijnKlasse.MethodeMK
BasisKlasse.MethodeBK
MijnKlasse.MethodeBK
BasisKlasse.MethodeBK
```

Op een analoge manier kunnen eigenschappen van klassen in afgeleide klassen overschreven worden.

De constructor van een afgeleide klasse moet steeds eerst de constructor van basis- of bovenliggende klasse oproepen. Het volgend voorbeeld illustreert hoe dit geïmplementeerd wordt.

```
public class BasisKlasse {
    public BasisKlasse() {
        System.Console.WriteLine("Constructor BasisKlasse");
    }
}

public class MijnKlasse:BasisKlasse {
    public MijnKlasse() : base() {
        System.Console.WriteLine("Constructor MijnKlasse");
    }
}
```

In de hoofding van de constructor van de afgeleide klasse *MijnKlasse* wordt de constructor van de basisklasse *BasisKlasse* aangegeven die opgeroepen moet worden voor het uitvoeren van de constructor van de afgeleide klasse. De constructor van de basisklasse kan uiteraard ook argumenten hebben.

### 1.12.2 Abstracte klassen

Het volgend voorbeeld illustreert de syntax voor abstracte klassen.

```
public abstract class AbstracteKlasse {
    public abstract void Methode();
}

public class MijnKlasse:AbstracteKlasse {
    public override void Methode() {
        System.Console.WriteLine("MijnKlasse.Methode");
    }
}
```

De uitvoer van de opdrachten

```
MijnKlasse mijnObject = new MijnKlasse();
mijnObject.Methode();
AbstracteKlasse absObject = mijnObject;
absObject.Methode();
```

is

```
MijnKlasse.Methode
MijnKlasse.Methode
```

## 1.13 Een eerste programma

### 1.13.1 De methode Main

Een applicatie moet tenminste één klasse hebben met de methode *Main*. Deze methode is het startpunt van de applicatie. Er zijn verschillende mogelijke signaturen voor de methode *Main*, onder andere de twee volgende.

```
public static void Main() {
    ... // uit te voeren opdrachten
}

public static void Main(string[] args) {
    ... // uit te voeren opdrachten
}
```

### 1.13.2 Compileren

Bestanden met C#-code hebben de extensie *cs*. Om een uitvoerbaar programma te maken heb je een bestand met C#-code nodig met één of meerdere klassen, waarvan juist één klasse met een *Main*-

methode. De standaard online-compiler wordt opgeroepen met het commando *csc*. Als *MijnBestand.cs* de code voor een C#-programma bevat, dan wordt om de volgende manier het uitvoerbaar programma *MijnBestand.exe* aangemaakt.

```
csc MijnBestand.cs
```

De naam van het uitvoerbaar programma is afgeleid van de naam van het cs-bestand en niet van de klasse met de *Main*-methode.

Verschillende bestanden compileren tot één programma gaat als volgt:

```
csc /out:Programma.exe Bestand1.cs Bestand2.cs Bestand3.cs
```

De optie *out* biedt de mogelijkheid om de naam van het uitvoerbaar programma te specifiëren.

### 1.13.3 Bibliotheek (Assembly)

Het is ook mogelijk om verschillende bestanden met klassen, ... te compileren tot een bibliotheek. De klassen uit deze bestanden hoeven niet tot dezelfde namespace te behoren. Het bestand dat de gecompileerde bibliotheek bevat heeft de extensie *dll*.

```
csc /target:library /out:Bib.dll Bestand1.cs Bestand2.cs
    Bestand3.cs
```

Wil je klassen, ... uit de aangemaakte bibliotheek gebruiken in een programma, dan moet je dit expliciet opgeven bij het compileren.

```
csc /reference:Bib.dll Programma.cs
```

### 1.13.4 .NET klasse-bibliotheken

Het .NET-platform bevat verschillende bibliotheken die bruikbaar zijn voor meerdere programmeertalen waaronder C#. In dit hoofdstuk maakten we onder andere gebruik van de namespaces *System* en *System.Collection*